Speeding Up Nek5000 with Autotuning and Specialization

Jaewook Shin,[†] Mary W. Hall,[‡] Jacqueline Chame,[§]
Chun Chen,[‡] Paul F. Fischer,[†] Paul D. Hovland[†]

[†]{jaewook,fischer,hovland}@mcs.anl.gov
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

[‡]{mhall,chunchen}@cs.utah.edu
School of Computing
University of Utah
Salt Lake City, UT 84112

[§]jchame@isi.edu
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292

ABSTRACT

Autotuning technology has emerged recently as a systematic process for evaluating alternative implementations of a computation, in order to select the best-performing solution for a particular architecture. Specialization optimizes code customized to a particular class of input data set. In this paper, we demonstrate how compiler-based autotuning that incorporates specialization for expected data set sizes of key computations can be used to speed up `Nek5000`, a spectral-element code. `Nek5000` makes heavy use of what are effectively Basic Linear Algebra Subroutine (BLAS) calls, but for very small matrices. Through autotuning and specialization, we can achieve significant performance gains over hand-tuned libraries (e.g., Goto, ATLAS, and ACML BLAS). Additional performance gains are obtained from using higher-level compiler optimizations that aggregate multiple BLAS calls. We demonstrate more than 2.2X performance gains on an Opteron over the original manually tuned implementation, and speedups of up to 1.26X on the entire application running on 256 nodes of the Cray XT5 `Jaguar` system at Oak Ridge.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Performance

Keywords

empirical performance tuning, autotuning, specialization

1. INTRODUCTION

The complexity and diversity of today's parallel architectures overly burden application programmers in porting and tuning their code. At the very high end, processor utilization is notoriously low, and the high cost of wasting these precious resources motivates application programmers to devote significant time and energy to tuning their codes.

To assist the application programmer in managing this complexity, researchers have devoted considerable effort in the past few years to autotuning software that uses empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance [4, 31, 11, 24, 27]. This paper focuses on a particular role for autotuning, used in conjunction with *specialization* for specific classes of known input sizes. Specialization information allows the autotuner to derive highly optimized, specialized versions of a computation for known input sizes. These specialized versions are then included in a customized library. At run time, the execution environment can invoke the appropriate specialized version from the library.

The approach presented in this paper is based on work by Tiwari et al. that applies compiler-based autotuning to computational kernels, demonstrating performance sometimes comparable to that of manually tuned codes [6, 8, 27]. Extending their approach to tuning whole applications, we apply it to `Nek5000` [1], a scalable spectral-element code. `Nek5000` was an early science application on Blue Gene/P at Argonne and was awarded 30,000,000 CPU-hours under INCITE in 2009 [2]. Its application areas include nuclear reactor modeling, astrophysics, climate modeling, combustion, and biofluids. The execution time of `Nek5000` is dominated by what are essentially matrix-matrix multiplies of small, rectangular matrices. These small matrices are of known sizes that remain the same for different problem sizes and different scales. While highly optimized BLAS libraries (vendor, ATLAS, Goto, etc.) are available, they are tuned for large, square matrices and do not perform as well for the small, nonsquare matrices used in `Nek5000`. Thus, specialization for small matrices is a particularly valuable optimization for `Nek5000`, but autotuning is needed to identify the best-performing implementation of each problem size and add it to the library. In addition to tuning individual BLAS calls, further performance gains are possible by aggregating multiple BLAS calls and then applying compiler optimizations to the resulting code; autotuning again identifies the best optimization strategy.

This paper describes how we combine autotuning with specialization to optimize `Nek5000`. Our target machine is the Cray XT5 `Jaguar` at Oak Ridge National Laboratory. We used CHiLL [8, 27, 14], a loop transformation framework based on the polyhedral model, to automatically generate specialized versions according to user-specified optimization strategies. CHiLL provides a high-level

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

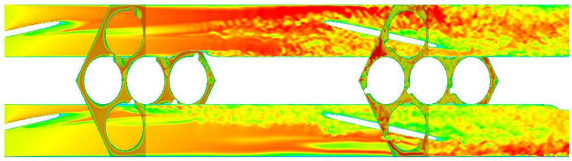


Figure 1: Turbulence in wire-wrapped subassemblies visualized by axial velocity distributions.

script interface that allows a user or compiler to specify an optimization strategy as a sequence of code transformations and parameters. We use this interface to describe the search space of specialized implementations. Using heuristics to prune the search space of possible implementations, a set of variants generated by CHiLL are then measured and compared, in order to derive the specialized library of implementations.

This paper makes several contributions:

- Automation of architecture-specific specialization for `Nek5000` using compiler tools. This process could be repeated to tune `Nek5000` for similar architectures and native compilers.
- Demonstration that compiler-based autotuning and specialization can exceed performance gains from library-based autotuning through the use of higher-level optimization that exploits application context; that is, how the libraries are invoked and the expected data sets.
- An optimization strategy that can be generalized for other applications that use matrix-matrix multiply on irregular-sized matrices for which BLAS libraries do not yield high performance or for related small, matrix operations for architectures supporting multimedia extensions such as SSE3. Other libraries, such as the widely used Portable, Extensible Toolkit for Scientific Computation (PETSc) library, would similarly benefit from automatic tuning specialized to application context.

Shin et al. [26] explored how specializing matrix multiply for small matrices could outperform hand-tuned libraries. This paper incorporates a detailed analyses based on dynamic instruction count information to capture the reasons behind the performance differences. More important, we have integrated a specialized library into the `Nek5000` application for a scaled-up data set size (`G6a`) and demonstrated up to 26% performance gains on 256 nodes of the `Jaguar` (Cray XT5) system at Oak Ridge, gains that are largely preserved as we increase the number of cores per node. Optimization at a higher level, where several of these matrix-matrix multiplications are combined and optimized together, yields an additional almost 3% performance gain.

2. NEK5000

`Nek5000` is a scalable code for simulating fluid flow, heat transfer, and magnetohydrodynamics as well as electromagnetics (in a separate code, NekCEM). The code is based on the spectral-element method (SEM) [21], a hybrid of spectral and finite-element methods. Spectral discretizations based on Nth-order tensor product polynomial expansions in $\hat{\Omega} := [-1,1]^d$, d=2 or 3, provide rapid convergence (high accuracy per grid point) at low cost. In particular, operators that are nominally full with $O(N^6)$ nonzeros can be applied in only $O(N^4)$ work with only $O(N^3)$ memory accesses. Moreover, the work can be cast as dense *matrix-matrix* products. The SEM extends the geometric flexibility of spectral methods in two ways. First, it uses Lagrangian interpolants based on Gauss or Gauss-Lobatto quadrature points that, in addition to ensuring stability, provide sufficiently accurate quadrature to allow pointwise evaluation of variable-coefficient integrands. Integrands involving Jacobians and metric tensors in deformed domains are thus evaluated with only N^3 storage and work complexity. In addition, the SEM allows multiple domains (deformable brick elements) to be coupled together in the traditional manner of finite elements to realize complex domain shapes. A recent example is the wire-wrapped rod-bundle flow of Figure 1, comprising 560,000 elements of order $N = 8$ [10].

The core computation in `Nek5000` calls for repeated function evaluations either for explicit substeps of the time advance or for iterations in implicit substeps. Within each element, each evaluation entails matrix-vector products of the form $C \otimes B \otimes A\underline{u}$. Specifically, we require sums of the following form.

$$v_{ijk} = \sum_{p=1}^{N} A_{ip} u_{pjk}, \qquad v_{ijk} = \sum_{p=1}^{N} B_{jp} u_{ipk},$$

$$v_{ijk} = \sum_{p=1}^{N} C_{kp} u_{ijp}, \qquad i,j,k \in \{1,...,N\}^3$$

The first product can be cast as a matrix multiply if u_{ijk} is viewed as an array having N^2 columns of length N. Similarly, the last product can be expressed as $V = UC^T$. The middle sum is expressed as a sequence of small products, $u(:,:,k)B^T$, $k = 1, \ldots, N$ [9]. Because the approximation order of the pressure and velocity spaces differ by 2, the above sums also appear with permutations in which index ranges may be replaced by $M = N - 2$. Thus, `Nek5000` requires numerous calls to small, dense matrix multiplies of known sizes over a limited range of values.

In this paper, we use `Nek5000` to illustrate our autotuning process and associated tools. We investigate the performance impact of autotuning and specialization for two `Nek5000` data sets: `Helix2`, which is helical pipe flow, similar to that found in certain vascular flows, and `G6a`, which is turbulent flow in a channel that is partially blocked by a cylinder.

3. COMPILER TECHNOLOGY: AUTOTUNING AND SPECIALIZATION

The methodology for optimizing `Nek5000` consists of three steps. We first use CHiLL to generate code versions specialized for specific matrix sizes. An automated empirical search then finds the best optimization parameters, using a set of compiler heuristics to keep the search space manageable. Finally, we create a library of specialized code versions and replace the original computation with calls to the library.

To put this approach in context, we have automated much of what is difficult in manual tuning – correct code generation, empirical measurements to explore a large set of implementations, pruning to avoid searching unprofitable implementations – but have relied on user involvement for some of the steps, particularly in describing the transformation strategies to be considered. This approach can evolve toward increased automation as tool technologies improve, but the interfaces will also always leave the door open for application developers to have control over optimization.

The tools described in this section, highlighted in Figure 2, are part of an autotuning workflow described elsewhere [19, 14]. This

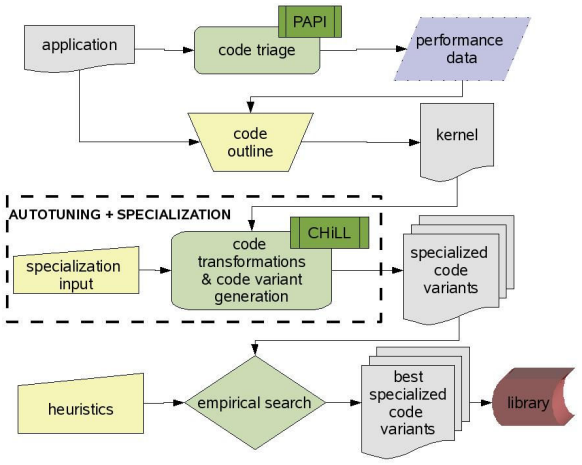


Figure 2: Overview of our approach.

workflow comprises several components. Code triage identifies computations that have optimization opportunities and performance issues. The programmer uses profiling or performance tools that correlate performance metrics with source code to identify computations to be tuned. Instrumentation (automatic or manual) might be used to collect performance-sensitive data, such as problem sizes, during execution of the application. Code outlining derives a standalone kernel for the key computations discovered in the triage process, along with the kernel's input data and parameter values collected during application runs. Once the bottlenecks of an application are identified and outlined into kernels, the extracted kernels need to be specifically tuned for the target architecture (autotuning and code generation).

3.1 Result of Code Triage for Nek5000

For `Nek5000` we used PAPI to collect hardware performance metrics and observed that, with the `Helix2` input, the application spends approximately 60% of the time on a particular function, `mxm44_0`. This function is a manually tuned implementation of matrix multiply, which yields overall good performance over a wide range of architectures. The main loop nest is unrolled by 4 for each of the `i` and `j` loops of the original loop nest shown in Figure 4(a). If either `M` or `N` is not a multiple of 4, clean-up loops execute the residual iterations.

We would like to improve on the manual optimizations of `mxm44_0` by automatically generating a library for the matrix sizes used for `Helix2` and `G6a` problems. To investigate the frequency of each array size, we instrumented `mxm44_0` so that it captures the number of calls for each matrix size across all of its invocations for each of the two problems. We use these call frequencies to select sizes for specialization and optimize the conditional checks for matrix size, as described in Section 3.5.

3.2 Optimization Strategy

Since the key computational kernel of `Nek5000` is matrix-matrix multiply, why is it not sufficient to use standard BLAS libraries for the target architecture? This point was the focus of Shin et al.'s paper [26], which compared various hand-tuned BLAS libraries to those generated by their compiler. For large, square matrices, such as 1024×1024, the ACML (native), ATLAS, and the Goto BLAS libraries all perform well, above 70% of peak performance. They incorporate aggressive memory hierarchy optimizations such as *data copy*, *tiling*, and *prefetching* to reduce memory traffic and hide memory latency. Additional code transformations improve *instruction-level parallelism* (ILP). Several examples in the literature describe this general approach [4, 31, 7, 12, 33].

If we look closer at matrices of size 10 or smaller, however, those same BLAS libraries perform below 25% of peak performance. Since these matrices fit within even small L1 caches, the focus of optimization should be on managing registers, exploiting ILP in its various forms, and reducing loop overhead. For these purposes, we can use *loop permutation* and aggressive *loop unrolling* for all loops in a nest. To the backend compiler, unrolling exposes opportunities for instruction scheduling, scalar replacement, and eliminating redundant computations. Loop permutation may enable the backend compiler to generate more efficient *single-instruction multiple-data* (SIMD) instructions by bringing a loop with unit stride access in memory to the innermost position, as required for utilization of multimedia-extension instruction set architectures.

Thus, in our initial set of experiments, we generate code using a combination of loop permutation and unroll-and-jam. In some cases, where the matrices are small, we obtain the best performance by coming close to fully unrolling all the three loops in the nest. When applied too aggressively, however, loop unrolling can generate code that exceeds the instruction cache or register file capacity. Therefore, we use autotuning to identify the unroll factors that navigate the tradeoff between increased ILP and exceeding capacity of the instruction cache and registers. We rely on the native backend compiler for the architecture to identify the SIMD instructions, and simply expose code to the backend that will be optimized most effectively.

Even better performance can be obtained by aggregating multiple calls to matrix-matrix multiply and optimizing the code to exploit reuse in registers and cache, as explained in Section 2. Being compiler-based, our approach can optimize the middle loop that contains multiple calls to matrix-matrix multiply. To do this, we inline the matrix-multiply function into the loop as shown in Figure 3. Then, the inlined loop nest is used as an input to Shin et al.'s autotuning framework.

```
do iz=1,10
   call mxm(A(1,1,iz),10,B, \
            10,C(1,1,iz),10)
enddo
```

(a) original

```
do iz=1,10
  do i=1,10
    do j=1,10
      C(i,j,iz) = 0.0d0
      do l=1,10
        C(i,j,iz) = C(i,j,iz) + A(i,l,iz)*B(l,j)
      enddo
    enddo
  enddo
enddo
```

(b) inlined

Figure 3: Function inlining for higher-level tuning.

3.3 Transformation Using CHiLL

One of the key challenges that a programmer faces during the tuning process is to try many different transformation strategies in order to find the best solution. This is an extremely slow and error-prone process. CHiLL provides a script interface to the programmer that can be used to apply complex loop transformation strategies on a loop nest by composing a series of loop transformations [6, 8, 27]. CHiLL's polyhedral framework provides a mathematical treatment of loop iteration spaces and array accesses, and transformation algorithms manipulate the polyhedral representations. The

transformations supported include data copying, tiling, index set splitting, loop permutation, unroll-and-jam, fission, fusion, and any unimodular transformation.

To use CHiLL for optimizing small matrix-multiply computations, consider the code in Figure 4(a). The matrix-multiply code is imperfectly nested because the C array is initialized to zero before multiplication. To permute the loop in j,k,i order, the programmer needs to specify only one permute transformation (Figure 4(c)); CHiLL generates the correct result, as shown in Figure 4(e). Further, the *unroll* command of CHiLL performs unroll-and-jam if the unrolled loop is an outer loop and if inner loops can be fused together legally. Figure 4(g) shows the result of all three unroll sizes u1, u2, and u3 set to 2. Such flexibility greatly helps programmers since they can now focus their effort on how the transformation affects the locality and the performance, instead of the details of generating the correct code.

For the purposes of *specialization*, we have added to CHiLL the *known* command, which allows the programmer to express known loop bounds. Within CHiLL, *known* adds additional conditions to the iteration spaces extracted from the original code. These conditions subsequently affect the quality of the generated code, permitting different specialized versions and determining, for example, whether unroll factors evenly divide loop bounds, so that the compiler can avoid generating cleanup code.

Figures 4(b), (c), and (d) show three sample CHiLL scripts used to generate three different loop orders in (a) (actual output is similar but with loop upper bounds fixed at 10), (e), and (f), respectively. For brevity, we show the simplest versions where all unroll amounts are 1. Loops are numbered starting from 1 for the outermost loop and increase as we move inward. In `permute`, the loop numbers are used to indicate the loop order after the permutation. The meaning of `unroll`*(stmt,loop,factor)* is to unroll the individual statement *stmt* (numbered from 0) within *loop* by unroll factor *factor* (shown as unbound variables). The three scripts shown in Figures 4(b), (c), and (d) differ in the number of `unroll` commands; this difference is necessary because the number of loops are different after loop permutation as shown in (a), (e), and (f) depending on the loop order given. These scripts plus additional ones for different loop orders combined with ten different `known` statements were used to generate the specialized BLAS library for `Nek5000`.

These CHiLL scripts, as well as the ones for higher-level optimization, are automatically generated. Given a loop order and an unroll factor for each loop, a `permute` command is generated with the loop order, and then an `unroll` command is generated for each loop for statement `s1` in Figure 4(a). If all loops in which statement `s0` is not enclosed are inside all loops in which it is enclosed, as in the loop order of 123, no other `unroll` commands are necessary. Otherwise, an `unroll` command should be generated for each loop in which statement `s1` is enclosed but is itself enclosed within the loop in which statement `s1` is not enclosed. For example, `i` and `j`-loops are such loops when the loop order is 312, that is, `kij` as in Figure 4(d). Since statement `s0` is not inside the `k`-loop but both `i` and `j`-loops are within `k`-loop after loop permutation, separate `unroll` commands are generated for statement `s0` for each of `i` and `j`-loop. This same approach can be applied to the higher-level loop nest in Figure 3(b). The only difference is that it is a 4-deep loop nest, and so four loops are considered for both the `permute` and `unroll` transformations.

3.4 Autotuning: Pruning the Search Space

With all the transformation scripts ready, the next step is to search for the best optimization parameters. To this end, we invoke CHiLL,

```
      do 10, i=1,M
        do 20, j=1,N
s0:       c(i,j) = 0.0d0
          do 30, k=1,K
s1:         c(i,j) = c(i,j) + a(i,k)*b(k,j)
30        continue
20      continue
10    continue
```

(a) `original.f`

```
permute([1,2,3])
known(M=N=K=10)
unroll(1,1,u1)
unroll(1,2,u2)
unroll(1,3,u3)
```

(b) Loop order i,j,k

```
permute([2,3,1])
known(M=N=K=10)
unroll(1,1,u1)
unroll(1,2,u2)
unroll(1,3,u3)
unroll(0,3,u3)
```

(c) Loop order j,k,i

```
permute([3,1,2])
known(M=N=K=10)
unroll(1,1,u1)
unroll(1,2,u2)
unroll(1,3,u3)
unroll(0,2,u2)
unroll(0,3,u3)
```

(d) Loop order k,i,j

```
  do 2, t2 = 1, 10, 1
    do 4, t6 = 1, 10, 1
      c(t6, t2) = 0.0d0
4     continue
    do 6, t4 = 1, 10, 1
      do 8, t6 = 1, 10, 1
        c(t6,t2)=c(t6,t2)+ \
                 a(t6,t4)*b(t4,t2)
8       continue
6     continue
2   continue
```

(e) After loop permutation in (c)

```
    do 2, t4 = 1, 10, 1
      do 4, t6 = 1, 10, 1
        c(t4, t6) = 0.0d0
4     continue
2   continue
    do 6, t2 = 1, 10, 1
      do 8, t4 = 1, 10, 1
        do 10, t6 = 1, 10, 1
          c(t4,t6)=c(t4,t6)+ \
                   a(t4,t2)*b(t2,t6)
10      continue
8     continue
6   continue
```

(f) After loop permutation in (d)

```
  do 2, t2 = 1, 9, 2
    do 4, t6 = 1, 9, 2
      c(t6, t2) = 0.0d0
      c(t6, t2+1,) = 0.0d0
      c(t6+1, t2) = 0.0d0
      c(t6+1, t2+1) = 0.0d0
4     continue
    do 6, t4 = 1, 9, 2
      do 8, t6 = 1, 9, 2
        c(t6, t2) = c(t6, t2) + a(t6, t4) * b(t4, t2)
        c(t6, t2+1) = c(t6, t2+1) + a(t6, t4) * b(t4, t2+1)
        c(t6, t2) = c(t6, t2) + a(t6, t4+1) * b(t4+1, t2)
        c(t6, t2+1) = c(t6, t2+1) + a(t6, t4+1) * b(t4+1, t2+1)
        c(t6+1, t2) = c(t6+1, t2) + a(t6+1, t4) * b(t4, t2)
        c(t6+1, t2+1) = c(t6+1, t2+1) + a(t6+1, t4) * b(t4, t2+1)
        c(t6+1, t2) = c(t6+1, t2) + a(t6+1, t4+1) * b(t4+1, t2)
        c(t6+1,t2+1) = c(t6+1,t2+1)+a(t6+1,t4+1)*b(t4+1,t2+1)
8       continue
6     continue
2   continue
```

(g) A complete example of script in (c) with u1=u2=u3=2

Figure 4: Example of CHiLL scripts and the generated codes.

in order to generate actual code variants and measure their performance on the target machine. For some of the smaller matrix sizes, the search space is sufficiently small that exhaustive search is feasible; but for the larger matrices, we must develop heuristics to prune the search space to complete the experiments. We extract the heuristics from the exhaustive search results of small matrices and use them in pruning the space of larger matrices. In this section, we briefly describe the pruning heuristics we used; a more detailed presentation is in [26].

Heuristic 1: Loop order. Loop orders that lead to lower-performance code variants are pruned from the search.

Heuristic 2: Instruction cache. Unroll amount for all three loops is limited by a constant C that is likely to fill the L1 instruction cache.

Heuristic 3: Unit stride on one loop. Unroll factor is restricted to those tuples (U_m, U_k, U_n), where at least one of U_m, U_k, or U_n is 1, to achieve spatial locality within a SIMD register.

Heuristic 4: Unroll factor divides iteration space evenly. When a loop of iteration count m is unrolled by a factor of u, the last "m mod u" iterations must be executed in a clean-up loop. The cost of executing the clean-up loop is significant when the matrices are small.

For the four-loop loop nest of Figure 3(b), exhaustive search is not possible even for the smallest input size. We use the last three heuristics, but now for a four-dimensional loop nest.

3.5 Building the Library

After each code variant is generated, it is compiled and linked with the driver that measures and records the performance. When the evaluation is complete for the generated code variants, the best-performing variant is selected for each matrix size, and these are aggregated into a library. The library includes a wrapper code that takes the same number of arguments and has the same name as the existing default implementation to provide the same interface to the rest of `Nek5000`, as in Figure 5. A specialized code is invoked if one is available that matches the three size parameters; otherwise, the original, manually tuned version is invoked.

To keep conditional check overhead reasonably low, we follow the algorithm shown in Figure 5(a). The algorithm takes three arguments: a set of input sizes (`InSet`), the number of dimensions of the elements (`d`), and a basic block (`BBlock`) in which codes are inserted. Each element of `InSet` is a vector representing an input size. For matrix multiplies, for example, an element of `InSet` is (10,10,10) for M, K, and N dimensions of the three arrays. We assume that each input size is associated with the call frequency we obtain by instrumenting `Nek5000`. We first pick from `InSet` an element `S` that is called most frequently. Then, we choose a dimension of `S` whose value is identical for the largest number of elements in `InSet`. Thus, we divide `InSet` into two disjoint subsets: one with the elements that have the same value for the chosen dimension and the other with the remaining elements. With this partitioning, we reduce the problem into two smaller problems. For the chosen subset, we generate an `if`-statement that checks for the chosen dimension with the common value of the set, and thus only `d`-1 dimensions remain to be checked. The algorithm recursively calls itself with `d`-1 for this subset, and once again for the remaining input sizes. For example, Figure 5(b) shows the wrapper code of the matrix multiply library for the `G6a` problem, which calls 17 specialized routines.

4. EXPERIMENTS

We tuned the library on a 2.5 GHz AMD Opteron Phenom workstation that has four cores. The machine has separate 64 KB L1 instruction and data caches, an integrated 512 KB L2 cache, 2

Algorithm `WrapGen(InSet`: set of vectors representing input sizes,
`d`: number of dimensions of a vector element in `InSet`, `BBlock`: basic block to generate code)

1. `S` ← Choose from `InSet` an element with the largest number of calls
2. `ThenSet`, i ← Find a subset `Sub` of `InSet` and i such that all elements of `Sub` has the same value as the i-the dimension value of `S` and |`Sub`| is the largest among such subsets
3. `IfStmt` ← Generate an **if**-statement in `BBlock` checking for the i-th dimension value
4. `RDimSet` ← Delete i-th dimension from all elements of `ThenSet`
   `WrapGen(RDimSet, d` − 1, **then**-block of `IfStmt)`
5. `WrapGen(InSet` − `ThenSet, d`, **else**-block of `IfStmt)`

(a) Algorithm

```
(1) mxm(a, m, b, k, c, n){
(2) if (all a, b and c are aligned to the SIMD register width){
(3)     if (k == 10){
(4)         if (m == 10){
(5)             if (n == 10){ m10_10_10(a,b,c); return;}
(6)             if (n == 100){ m10_10_100(a,b,c); return;}}
(7)         else if (n == 2){
(8)             if (m == 2){ m2_10_2(a,b,c); return;}
(9)             if (m == 4){ m4_10_2(a,b,c); return;}}
(10)        else if (m == 100 && n == 10){m100_10_10(a,b,c); return;}
(11)        else if (n == 16){
(12)            if (m == 16){ m16_10_16(a,b,c); return;}
(13)            if (m == 256){ m256_10_16(a,b,c); return;}}
(14)        else if (m == 16 && n == 100){m16_10_100(a,b,c); return;}}
(15)    else if (k == 2){
(16)        if (n == 10){
(17)            if (m == 10){ m10_2_10(a,b,c); return;}
(18)            if (m == 100){ m100_2_10(a,b,c); return;}}
(19)        else if (m == 10 && n == 88){m10_2_88(a,b,c); return;}}
(20)    else if (k == 16){
(21)        if (m == 16){
(22)            if (n == 16){ m16_16_16(a,b,c); return;}
(23)            if (n == 256){ m16_16_256(a,b,c); return;}}
(24)        if (m == 10){
(25)            if (n == 10){ m10_16_10(a,b,c); return;}
(26)            if (n == 256){ m10_16_256(a,b,c); return;}}
(27)        else if (m == 256 && n == 16){m256_16_16(a,b,c); return;}
(28)        else if (m == 100 && n == 10){m100_16_10(a,b,c); return;}}}
(29) mxm44_0(a, m, b, k, c, n);}
```

(b) Example wrapper for matrix multiplies of `G6a`

Figure 5: Wrapper code generation for specialized routines.

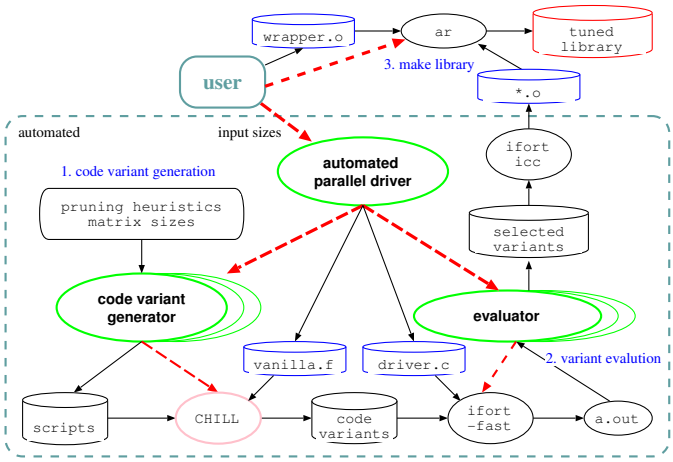
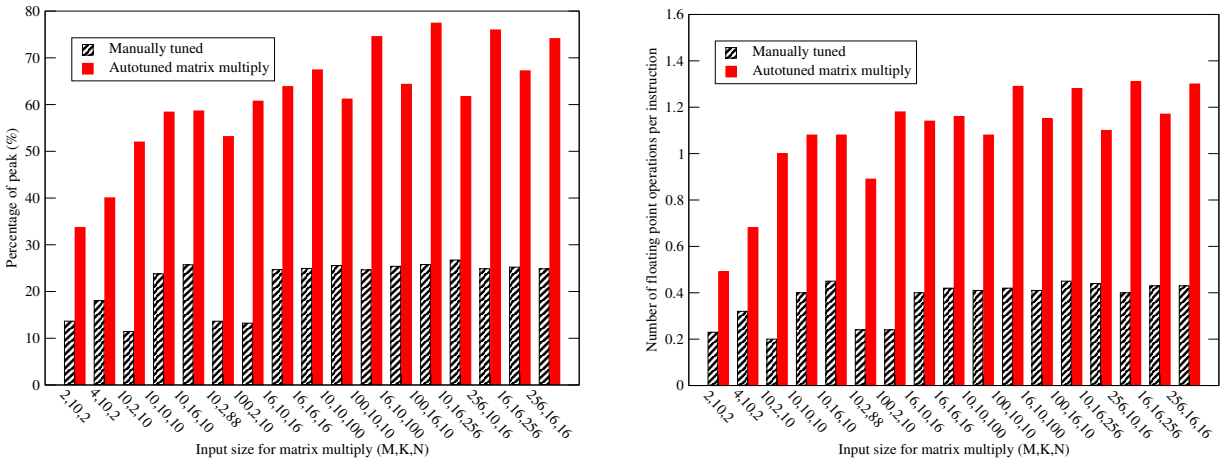


Figure 6: Block diagram describing autotuning tools and experiment.



(a) Percentage of machine's peak

(b) Floating-point operations per instruction

Figure 7: Performance of the matrix multiply library for 17 input sizes chosen for `G6a`.

MB L3 cache, and 6 GB of memory. Since it runs 64-bit Linux (`Ubuntu_8.04-x86_64`), all 16 XMM registers are available for use. In order to allow an access to hardware performance counters, the Linux kernel (version 2.6.25.4) is patched with `perfctr` distributed in PAPI version 3.6.0 [5]. CHiLL version 0.1.5 [8] and the Intel compiler version 10.1 [17] are used to transform and compile the code variants. We measured the whole `Nek5000` performance on `Jaguar` at Oak Ridge National Laboratory, a Cray XT5 supercomputer with new six-core Opteron Istanbul processors. The architecture is similar to Phenom, but there are 6 cores per socket, 2 sockets per node, 16 GB of memory per node, and each core runs at 2.6 GHz. The compiler we use is PGI `pgcc` and `pgf90` version 9.0-4, which is the default compiler on the machine.

Figure 6 shows a block diagram of the experimental flow. Given input matrix sizes for which specialization is desired, a matrix-multiplication kernel (`vanilla.f`), shown in Figure 4(a), and a driver (`driver.c`) are used to measure and collect the performance of code variants. The driver measures the number of clock cycles using PAPI performance counter PAPI_TOT_CYC. To obtain accurate measurements, we execute a variant 500 times per measurement for matrix multiply and 10 times per measurement for the higher-level kernel. We collect 100 such measurements and record the minimum as the final performance of the variant. We produce as output a high-performance library of specialized matrix-multiplication routines. Parameters m, k, and n are used in defining matrix sizes as in $C(m,n) = A(m,k) \times B(k,n)$. Code variants are generated in Fortran, but the driver is a C function. In order to generate aligned SIMD instructions, an `__attribute__((aligned (16)))` qualifier is added to array declarations, and the interprocedural optimization feature is used with `-fast` for `ifort` and `-O3 -ipo` for `icc`. Currently, all components inside the dotted box of Figure 6 are automated. Also, *automated parallel driver* can invoke multiple copies of *code variant generator* and *evaluator* to exploit task parallelism when multiple cores are available.

We optimize `Nek5000` for the `Helix2` and `G6a` input data sets. We chose the fastest variant for each input size to create two libraries, one for `Helix2` and another for `G6a`. Table 1 summarizes the number of input sizes for two versions for each of the two problems. At first glance, 18 specialized routines might seem too many; but because of the call frequency distribution and the way we group and order the conditional checks, at most 5 additional checks are needed beyond the required checks for the three values. Furthermore, this worst-case overhead occurs for less than 1% of the calls because the conditional checks are ordered in decreasing order of the call frequencies as described in Section 3.5. The tuned library covers close to 99% of computation originally performed in `mxm44_0`.

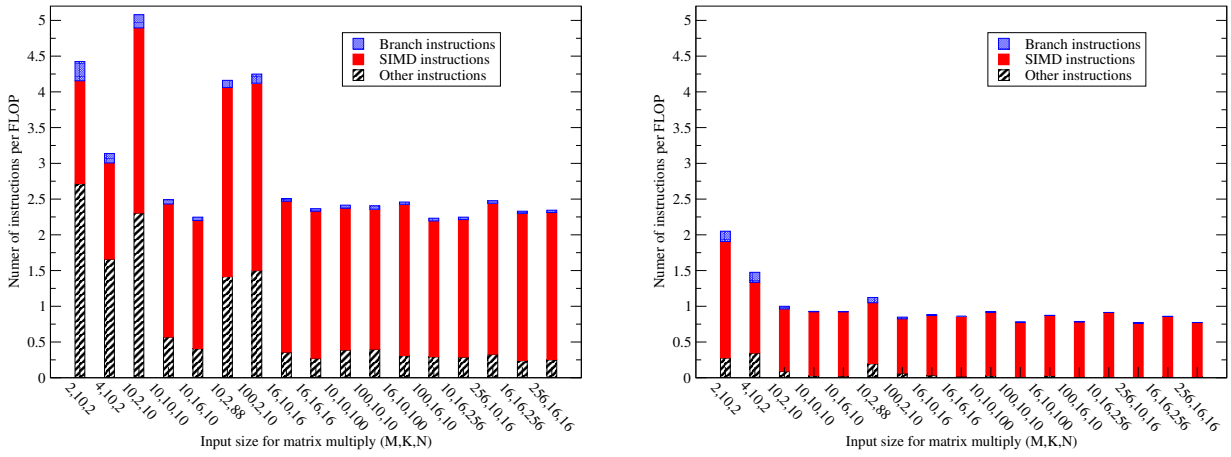
Table 1: Number of specialized routines for tuned libraries.

Libraries	`Helix2`	`G6a`
Autotuned matrix multiply	18	17
Autotuned matrix multiply + higher-level kernels	15 (matrix multiply) 6 (higher-level)	12 (matrix multiply) 6 (higher-level)

4.1 Performance of Matrix-Multiply Kernels

What is the impact of using the specialized libraries as compared to simply using highly tuned BLAS libraries for the Opteron? In fact, the use of specialization, autotuning, and optimizations focused on small matrices yields significant gains over even the best manually tuned libraries. Shin et al. compared the performance difference between autotuned, specialized libraries and hand-tuned BLAS libraries [26]. The manually tuned libraries included the native ACML BLAS (version 4.1.0), Goto BLAS (goto_barcelona-r1.26), and ATLAS (version 3.8.2 with the following architectural default: AMD64K10h64SSE3). Each performs less than 30% of peak for the smallest matrices, which is more than 3X faster than using the native compiler on a naive implementation. The manually tuned baseline, part of `Nek5000`, is a little slower than the hand-tuned libraries, performing at roughly 23% of peak across all sizes.

In this paper, we go beyond the prior work of Shin et al. to compare the results of our automatically generated library to the baseline, which is near that of the manually tuned libraries. As shown in Figure 7(a), our automatically generated library code yields performance up to 77% of peak, more than a 2.2X improvement over the manually tuned code. To investigate the source of the speedups, we used PAPI to gather dynamic measurements of total number of instructions, branch instructions (PAPI_BR_INS), and SIMD instructions (PAPI_VEC_INS). Figure 7(b) shows the number of floating-point operations per instruction. Since AMD Opteron processors can compute at most two double-precision floating-point operations per instruction, the maximum value is 2. The shapes of



(a) Baseline (`mxm44_0`)

(b) Tuned kernels

Figure 8: Normalized instruction breakdown for Figure 7.

the two graphs of Figure 7 are similar: the y-axis of (b) goes up to 1.6, which mirrors 80% of peak in the y-axis of (a). This similarity suggests that the key performance difference is the number of instructions executed. Figures 8(a) and 8(b) show a more detailed analysis of the breakdown of instructions, normalized by the number of effective floating-point operations. Compared to the baseline, on average the tuned kernels have 6.5X fewer branch instructions, because of aggressive loop unrolling. Even though branches make up a small percentage of the total instructions, they can be costly because of small loop trip counts interfering with branch prediction. The specialized library code has 2.4X fewer SIMD instructions. However, the SIMD instructions make up a higher percentage of instructions in the specialized library, more than 80% of all instructions. Note that some instructions using SIMD functional units such as `mulsd` are counted as SIMD instructions, but they compute a single value. For example, the baseline version has many `mulsd`'s that compute a single value, whereas the tuned version has only `mulpd`'s that compute two values. This result shows that efficient SIMD code generation is a deciding factor for achieving high performance in dense matrix multiply of small matrices. Resource stalls (PAPI_RES_STL) per instruction were not significantly different for the two versions. All other instructions are reduced by 20X, because of redundancy elimination and more effective use of SIMD instructions.

4.2 Performance of Higher-Level Kernels

Figure 9 shows the additional performance gain from the tuned versions of the higher-level kernels, shown in Figure 3(b), for six input sizes of `G6a` on a Phenom processor (the first three bars) and on a Cray XT5 supercomputer (the last bar). The higher-level kernels optimize a collection of calls to matrix multiply. The x-axis of the graph represents different matrix sizes corresponding to the four loop bounds of `iz`, `i`, `l` and `j`-loops; the y-axis captures percentage of machine's peak. For each matrix size, the first bar provides the performance of the manually tuned baseline `mxm44_0`, between 11% and 26% of peak. The second bar, labeled `Autotuned matrix multiply`, represents the optimized matrix-multiply library from the previous section (without higher-level kernels), with a performance between 28% and 66% of peak.

The third bar, labeled `Autotuned higher-level`, represents the higher-level kernels, where we tune the outer `iz`-loop together with the dense matrix-multiply loop nest. The library

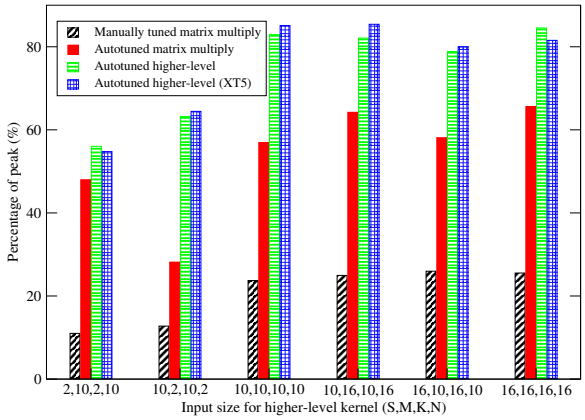


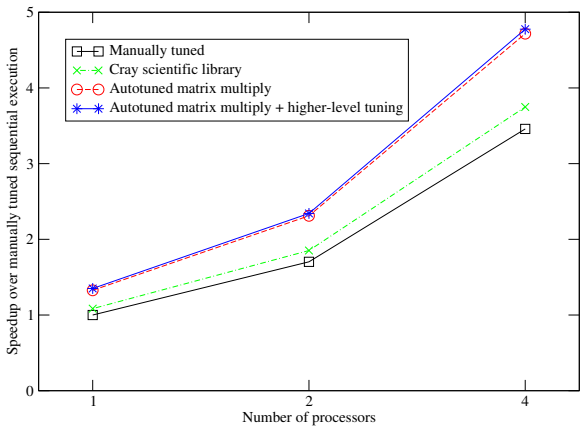
Figure 9: Performance of the tuned higher-level kernel of Figure 3(b).

achieves a much higher performance, between 56% and 84% of peak.

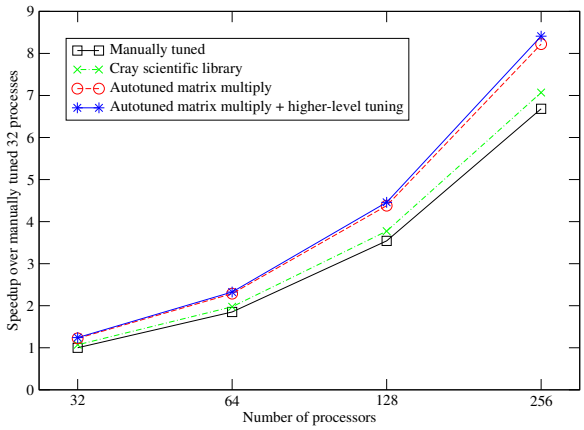
As preparation for using the tuned library on a large parallel machine, the binary library was transferred to `Jaguar` at Oak Ridge National Laboratory. (Note that it is not feasible to tune the library on the Oak Ridge system because of usage restrictions on login nodes and lack of a compiler on compute nodes.) We compare performance to see whether the binary library tuned on the Phenom platform will still achieve high performance on `Jaguar`. We discover that the results on the Cray XT5 are comparable, making the use of this tuned library performance-portable across these related but different platforms.

4.3 Performance of Nek5000 on Jaguar

In this section, we present the performance gains of the whole `Nek5000` achieved by the autotuned libraries for a small data set `Helix2` and a larger data set `G6a`. We compare four versions of the application representing different implementations of matrix multiply: (1) manually tuned `mxm44_0`; (2) Cray scientific library 10.4.0 for Istanbul, which uses the Goto BLAS (CSL/Goto); (3) the tuned library of specialized matrix multiplies; and (4) the tuned



(a) `Helix2` on `Jaguar` using 1 core per node



(b) `G6a` on `Jaguar` using 1 core per node

Figure 10: Speedups of `Nek5000` by varying number of nodes.

library that also includes higher-level kernels. For this experiment, we used the `time` tool in Linux to measure the wall clock time on the `Jaguar` supercomputer. We ran the whole program 15 times for 300 time steps for `Helix2` and 1,000 time steps for `G6a` and took the minimum run time, which reduces the effects of large I/O performance fluctuations.

Figures 10(a) and 10(b) show the result of this experiment on `Helix2` for 1 to 4 processors and `G6a` for 32 to 256 processors, respectively. The range of number of processors is problem dependent: `G6a` cannot run on fewer than 32 processors, while `Helix2` is too small to scale beyond 4 processors. For this initial experiment, we used a single core per Istanbul node, so the number of cores corresponds to number of nodes. The speedups of the graph are computed with respect to the baseline version that runs on a single processor for `Helix2` and on 32 processors for `G6a`. In general, as the number of processors is doubled, the performance improves by a factor of 1.7 to 2. When compared with the manually tuned versions running on the same number of processors, the versions using the CSL/Goto library speed up by 6% and 9%, and the versions using the autotuned matrix multiply kernels by 32% to 36% for `Helix2` and around 23% for `G6a`. When the autotuned higher-level kernels are used in addition, we gain an additional 2%; and the overall gains over the manually tuned versions on the same number of processors are between 35% and 38% for `Helix2` and up to 26% for `G6a`.

Now we examine how performance gains are impacted by using multiple cores per node. Figure 11 presents performance of `G6a` on 32 nodes, using between 1 and 12 cores per node, as speedup over the manually tuned baseline running on 32 nodes. Performance increases until 8 cores per node, that is, a total of 256 cores, before it drops a little. We believe the drop from 8 to 12 cores per node is due to memory-related issues, such as memory latency, bandwidth saturation, or competition for shared L3 cache, or it is related to the shared-memory implementation of MPI. Note that it is not due to lack of available parallelism, as the 256-node single-core results from Figure 10(b) show significantly better performance (46% better) than the equivalent computation decomposition of 32-node 8-core results. Nevertheless, for the same number of cores per node, the autotuned version is always better than the manually tuned version and is still roughly 12% faster for 8 cores per node.

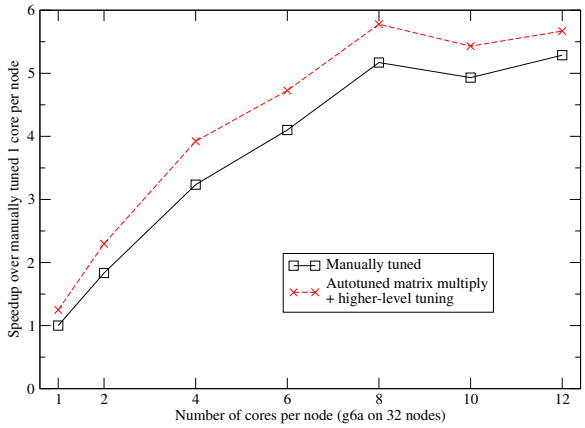


Figure 11: Speedups of `Nek5000` by varying the number of cores per node.

5. RELATED WORK

Library-based autotuning has been successful for dense and sparse linear algebra [4, 31, 30] and signal processing [11, 24, 29]. OSKI [30] tunes sparse matrix computation automatically. These library-based systems are able to autotune for a particular hardware, but they tune only a fixed set of library kernels and are not able to tune arbitrary computations.

Compiler-based autotuning systems and code transformation tools have also been actively studied recently. Chen et al. [7] combine compiler models and heuristics with guided empirical evaluations to take advantage of their complementary strengths. Tiwari et al. [27] use the Nelder-Mead algorithm in their autotuning system that combines Active Harmony and CHiLL. Hartono et al. [15] use annotations to describe performance improving code transformations. POET is a scripting language for parameterizing complex code transformations [32]. Pouchet et al. [22, 23] embed legality of affine transformations as linear constraints, thereby combining the code transformation steps and the legality checking step. Kulkani et al. [18] describe VISTA, which allows selecting the order and scope of optimization phases in compiler.

Herrero and Navarro [16] describe specializing matrix multiplication for small matrices. However, their code variants were generated manually. For tuning `Nek5000`, the most closely related is Paul Fischer et al.'s work on scaling `Nek5000` on Blue Gene [10]

and ASCI Red [28]. Our approach improves single-node performance based on compiler technology. Gunnels et al. [13] provide strategies for blocking matrices for matrix multiplication at each level of hierarchical memories. However, these do not apply to the small matrices we encounter in `Nek5000`. Barthou et al. [3] reduce the search space by separating optimizations for in-cache computation kernels from those for memory hierarchy. Similar to our approach, they obtain high-performance kernels from the vendor compiler by providing it with simplified kernels. To generate code variants, they use X Language controlled by user-provided pragmas. Shin et al. [26] describe a compiler-based technique that combines *specialization* with autotuning for matrix multiply of small, rectangular matrices. In this paper, we demonstrate that the same autotuning strategy can be used to tune higher-level kernels that are application specific. Further, we automate the strategy in an autotuning system that can exploit multiple cores in a system, and we demonstrate that single-node performance gains can improve scaled-up supercomputer performance.

As a design choice, we could have generated SIMD instructions directly from our tool chain using SIMD intrinsics [24, 20, 25]. This would allow us to have finer control over SIMD code generation. Instead, we have the backend compiler to perform SIMD parallelization by providing it with dependence and alignment information. With this choice, our tool chain became simpler, faster, and depends less on any particular architecture, still achieving up to 84% of machine's peak.

6. CONCLUSION

This paper described an autotuning and specialization methodology applied to `Nek5000`. The tuning process involves identifying data set sizes for the core computation (matrix multiply of small matrices) and providing a set of parameterized optimization scripts to the CHiLL polyhedral framework that generate specialized code. A set of heuristics prune the space of parameter values and variants as part of autotuning the implementation. We show speedups of more than 2.2X on the core computation as compared to already manually tuned code, and also much better performance than standard BLAS libraries. Performance improvements for the full `Nek5000` application are 38% on 4 nodes for `Helix2` and up to 26% on 256 nodes for `G6a`. These performance gains from the tuned library are significant – increased efficiency of a production code running on a supercomputer increases the precision of a result that can be run in the same time, or increases the throughput of `Jaguar`, an important shared national resource.[1]

Beyond tuning `Nek5000`, this paper shows an approach to tuning code that is repeatable and permits the application to maintain high-level, architecture-independent code. We can view the CHiLL script, search-space pruning heuristics and generated library as documentation of the tuning process for a particular platform and native compiler. Over time, as this code is tuned for multiple architectures, this prior tuning work can potentially be reused, thus systematically evolving application code for new architectures from machine-independent code.

Acknowledgments.

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357 and DE-SC00003777. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

[1] As a measure of the significance of this particular code, `Nek5000` has been allocated 30 million CPU-hours under the INCITE program.

7. REFERENCES

[1] http://nek5000.mcs.anl.gov.

[2] 2009 INCITE Fact Sheet, 2009. http://www.sc.doe.gov/ascr/incite/index.html.

[3] Denis Barthou, Sebastien Donadio, Alexandre Duchateau, William Jalby, and Eric Courtois. Iterative compilation by exploration of kernel decomposition. In *The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, New Orleans, LA, 2006.

[4] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, pages 340–347, Vienna, Austria, 1997.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[6] Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.

[7] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.

[8] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, Computer Science Department, 2008.

[9] M.O. Deville, P.F. Fischer, and E.H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge, 2002.

[10] Paul Fischer, James Lottes, David Pointer, and Andrew Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125, 2008.

[11] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the West. Technical Report MIT-LCS-TR728, MIT Lab for Computer Science, 1997.

[12] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. Van De Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[13] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. High-performance matrix multiplication algorithms for architectures with hierarchical memories. Technical Report CS-TR-01-22, University of Texas at Austin, 2001.

[14] Mary W. Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, and Gabe Rudy. Transformation recipes for code generation and auto-tuning. In *International Workshop on Languages and Compilers for Parallel Computing*, October 2009.

[15] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009.

[16] José R. Herrero and Juan J. Navarro. Improving performance of hypermatrix Cholesky factorization. In *9th International Euro-Par Conference*, pages 461–469, 2003.

[17] Intel. *Intel Fortran Compiler User and Reference Guides*,

2008. http://www.intel.com/cd/software/products/asmona/eng/406088.htm.

[18] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, San Diego, CA, 2003.

[19] Chunhua Liao and Dan Quinlan. A Rose-based end-to-end empirical tuning system for whole applications. Technical Report UCRL-SM-210032-DRAFT, Lawrence Livermore National Laboratory, August 2009.

[20] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization - revisited for short SIMD architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 2–11, 2008.

[21] A. T. Patera. A spectral element method for fluid dynamics - laminar flow in a channel expansion. *Journal of Computational Physics*, 54:468–488, 1984.

[22] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Fifth International Symposium on Code Generation and Optimization (CGO'07)*, San Jose, CA, 2007.

[23] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, AZ, 2008.

[24] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[25] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Exploiting superword-level locality in multimedia extension architectures. *Journal of Instruction Level Parallelism (JILP)*, 5:1–28, 2003.

[26] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, and Paul D. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *The Fourth International Workshop on Automatic Performance Tuning*, October 2009.

[27] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *IPDPS*, Rome, Italy, May 2009.

[28] Henry M. Tufo and Paul F. Fischer. Terascale spectral element algorithms and implementations. In *ACM/IEEE conference on Supercomputing*, Portland, OR, 1999.

[29] Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, Seattle, WA, 2009.

[30] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 2005.

[31] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing*, 1998.

[32] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *IPDPS*, Long Beach, CA, March 2007.

[33] Kamen Yotov, Xiaoming Li, Gang Ren, María Jesús Garzarán, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.